On Improving Constrained Single and Group Operator Placement Using Evictions in Big Data Environments

Nikos Tziritas, Thanasis Loukopoulos, Samee U. Khan, Senior Member, IEEE; Cheng-Zhong Xu, Fellow, IEEE; Albert Y. Zomaya, Fellow, IEEE

Abstract—With an ever increasing amount of data generated by scientific experiments, social networks and mobile as well as wireless sensor networks, reducing resource consumption by big data applications becomes of paramount importance. Towards this end, filtering data close to the data sources is a common strategy in order to reduce network traffic. Assuming a network of nodes, each potentially generating data and a query in the form of a single operator to be applied in these data, the basic statement of the operator placement problem is: find the best node to place the operator so that the network traffic is minimized. In this paper we study the problem of placing a set of communicating operators exhibiting a tree structure over a tree network of nodes with capacity constraints. We take advantage of our previous work on unconstrained placement in order to develop a new approach enabling both single and group operator migrations using evictions of hosted operators if free space is required. To enhance their applicability, the algorithms work in a distributed asynchronous manner, requiring only minimal knowledge at each network node. Results from simulation experiments show that the proposed algorithms reduce considerably network overhead against their counterparts.

Index Terms—single and group operator migrations; constrained placement; distributed algorithms; evictions; mobile big data

1 INTRODUCTION

Shifting data intensive computations close to data sources is a standard strategy to alleviate excessive network load in a Big Data environment. Of particular interest is the case of applications accessing, filtering and combing data from multiple sources. Such applications can be encoded as a single or a set of operators to be applied over data streams.

Consider the example of Fig. 1 depicting a generic application that detects the involved persons in people abandoning after car accidents. The application is structured as a set of the following four operators: (i) a video processing operator taking input from street camera feeds and detecting abnormal situations with image processing techniques; (ii) a social network data operator tasked with mining social networks to discover potential eye witnesses; (iii) a mobile data operator tasked with identifying people that had been near the accident scene by analyzing digital footprints in cellular base stations and (iv) a monitoring operator that coordinates the actions of the above three operators, accumulates and filters data from them and sends notifications to a sink, e.g., local authorities.

Clearly, if an operator processes data residing in a node other than the one hosting the operator, these data must be transferred. The basic form of the operator placement problem can be stated as: given a network of N nodes with some or all of them generating data processed by an operator, place the operator in the network so as to reduce the network traffic.

The problem was coined from the database field in the context of in-network query processing ([10] and [11]) with the optimization goal being response time rather than network load and has found renewed interest with
the advent of sensor networks and big data applications [1] with the primary optimization objective being the network traffic. In this paper we tackle the problem of placing a set of operators logically forming a tree (operator-tree) over a tree network. This scenario directly matches the case of query placement over a WSN [7], since query plans usually have a tree structure, but also exhibits broader scope in the cases where the network can be approximated by a spanning tree. In particular, we investigate the case of constrained placement whereby the nodes of the network can hold only a portion of the operators due to capacity constraints such as processing power.

Our contributions boil down to the following: (a) we propose an approach enabling both single and group operator migrations using evictions of hosted operators if free space is required; (b) we prove that our problem is NP-complete in the case of resource-constrained nodes, rendering in that way the decision of which operators must be evicted to host a beneficial (in terms of network overhead) group of operators not trivial at all; (c) Our approach handles in a sophisticated way of how we can judiciously decide a group of operators to be evicted from a reference node such that to free space; (d) The proposed algorithm provably converges and requires minimum knowledge from nodes; and (e) we conducted experiments showing that our approach outperform other existing proposals in the literature with the performance difference ranging between 5% and 65% depending on the particular test case.

We must stress that the problem addressed in this paper is orthogonal to the problem of reducing the processing time by exploiting parallel and processing techniques like “map-reduce”.

The rest of the paper is structured as follows. Section 2 contains related work; Section 3 describes the system model and formulates the problem; Section 4 and 5 present the eviction-based algorithms considering single and group operator migrations, respectively; convergence issues are discussed in Section 6; Section 7 provides an experimental evaluation through simulation; finally, Section 8 concludes the paper.

2 RELATED WORK

The operator placement problem has been extensively studied in the past decade. The optimal placement of a single operator within the system has been tackled by [1]. The work in [7] tackles the same problem considering multiple operators within the system. Specifically, the operators are structured as a binary tree, and the problem is solved in an optimal way given that more than 50% compression is achieved at each operator. A flooding algorithm is proposed in [18] to tackle the operator placement problem for general-structured applications. The solution results always in the optimal placement at the expense of flooding all of the nodes within the system with control messages. On the other extreme, [8] addresses the problem under the objective of optimizing the time experienced by users. Unfortunately, all of the aforementioned approaches consider only unlimited processing resources on the nodes within the system. Evidently, such approaches are not suitable for big data environments where the processing demands of operators may be huge due to the vast amount of data needed to be processed. A complementary work to ours is that of [4] that makes decisions about the placement of data sources as well as their replication.

The operator placement problem can also be thought of as the agent placement problem in WSNs under the following transformation. In the context of agent placement an application consists of mobile and immobile agents, with the first ones performing general-purpose processing, while the latter ones executing node-specific operations [12]. Mobile agents can be interpreted as operators, while immobile agents can be interpreted as data sources (a sink can also be thought of as a data source). There exist many mobile code based systems ([5], [6], [12], to name a few).

In the context of agent placement in WSNs, centralized algorithms for jointly optimizing network load and total storage space within the system were discussed in [16]; while fully distributed algorithms for optimizing network overhead were discussed in [13], [14], and [15].

Virtual machine (VM) placement [2] is also a problem quite close to the operator placement problem. Specifically, each VM accesses some data located on the underlying network. VMs play the role of operators. The VM placement problem is addressed in [3], whereby the objective is to minimize the network congestion within the system. The same problem is also tackled in [17] to optimize slowdown. The dynamic service placement problem is tackled in [19], with the objective being to reduce the hosting cost over time according to both demand and resource price fluctuation. A fully distributed algorithm is proposed in [9], called DBA, to solve the same problem tackled in this paper under the context of clouds. DBA works for general-structured graphs and takes into account capacity constraints on nodes. The difference with our approach is that DBA does not consider migrating group of VMs, resulting in that way in sub-optimal placements.

Perhaps the closest to our work are [13] and [15]. Specifically, [13] uses group migrations to achieve optimal placement in the unconstrained case, but tackles the constrained case with a rather simple capacity reservation approach. In [15] a more sophisticated capacity reservation approach was used allowing the evictions of hosted operators if free space is required. Nevertheless, the technique was introduced for single operator migrations which are suboptimal in the unconstrained case.

3 APPLICATION, SYSTEM MODEL AND PROBLEM FORMULATION

We consider applications structured as hierarchies (trees) of mobile software components, called operators. An operator takes input data from either other operators or directly from data sources. Operators implement higher-level data aggregation or processing functionality. This is
done using general-purpose computing resources. Once an operator is created, the system may decide at runtime to move it to another node.

### 3.1 System Model and Problem Formulation

The system consists of nodes with special sensing/actuating capabilities as well as general-purpose processing nodes. Let $n_i$ and $c(n_i)$ denote the $i$th node and its processing capacity, respectively. We assume a tree-based routing structure, whereby any two nodes are connected via a single, possibly multi-hop, path. Let $r_{ij}$ denote the number of hops between $n_i$ and $n_j$. We assume that the links of the routing structure are bidirectional, thus $r_{ij} = r_{ji}$. Also, $r_{ii} = 0$.

The system can host several applications, each one having its own operators. Let $o_k$, $s(o_k)$, $h(o_k)$ be the $k$th operator in the system, its processing demands and the node hosting it, where $1 \leq k \leq O$. $O$ enumerates all operators within the system. The $i$-th data source is denoted by $d_i$, with $1 \leq i \leq D$ enumerating all data sources within the system. The $j$-th sink is denoted by $s_j$, with $1 \leq j \leq S$ enumerating all sinks within the system. Also, let $T$ be a $(O + D + S) \times (O + D + S)$ matrix that encodes the data dependencies between data sources, operators, and sinks. Specifically, $T_{xy}$ denotes the unidirectional data demands from the abstract object $a_x$ to $a_y$, i.e., the number of data units $a_x$ sends to $a_y$ over a specific period (note that, in the general case, $T_{xy} \neq T_{yx}$). Where $a_x$ denotes the $x$-th abstract object: (i) operator when $1 \leq x \leq O$, (ii) a data source when $0 + 1 \leq x \leq O + D$, and a sink when $0 + D + 1 \leq x \leq O + D + S$. An abstract object $a_x$ may exchange data with its relatives (parent or children) in the application tree, let these relatives be denoted by $R_S$.

The objective of this paper is to reduce the amount of data exchanged between nodes due to the application-level communication. Without loss of generality, we assume the operators of an application are placed on the nodes of the system in a non-optimal way. Then, our goal is to perform a series of operator migrations to achieve a better operator placement reducing (ideally, minimizing) the total network traffic.

### 3.2 Migration benefit/penalty and eligibility

We focus on distributed solutions whereby each node decides locally which operator to migrate on which node, based on their data dependencies with other operators, sinks, and data sources.

Using the previous notations, the load incurred by $o_k$ when placed at $n_i$ can be expressed by Eq. (1). Let $M_{ij}^k$ refer to the migration of $o_k$ from $n_i$ to $n_j$. The benefit/penalty of $M_{ij}^k$, in terms of the load difference (positive or negative) of the placement obtained after $M_{ij}^k$ takes place compared to the current placement, is given by Eq. (2).

In order for $M_{ij}^k$ to be eligible, $n_j$ should have enough free processing capacity Eq. (3). Each migration leads to a new placement that may incur a lower or perhaps a higher operator-level communication over the network. In the former case, we refer to the migration as beneficial else non-beneficial. Not all beneficial migrations are eligible, due to the capacity constraint Eq. (3).

### 3.3 Evictions

To alleviate the aforementioned problem, we consider performing (possibly non-beneficial) migrations that free node processing capacity. We refer to such migrations as evictions. The idea is to exploit the processing capacity being released such that to perform beneficial migrations. Obviously, per definition, evictions cannot (by themselves) reduce the amount of application-level traffic over the network. To achieve such a load reduction evictions must be followed by at least one migration with a benefit that outweighs their penalty.

In the sequel, we give an example to illustrate this scenario. Assume the application depicted in Fig. 2, that comprises three data sources ($d_1, d_2, d_3$), two operators ($o_1, o_2$), and one sink ($s_1$). The link weights represent the data dependencies among operators/data sources/sink. Assume that the application is deployed in a network of seven nodes as shown in Fig. 3, where each node has enough processing capacity to host only one operator.

\[
l^E = \sum_{a_y \in R_S} (T_{xy} + T_{yx}) r_{ih(a_y)}
\]

\[
B^k_{ij} = l^k - c(n_j) \geq \sum_{o_m} (o_m) h(a_m) = n_j
\]

We let first consider the operator $o_1$. There is no better placement for it, because every migration of $o_1$ away from $n_1$ is non-beneficial as per Eq. (2). Let us now consider operator $o_2$. In this case, a migration from $n_4$ to $n_1$ would yield a benefit of 9 as per Eq. (2). But note that $M_{41}^2$ is not feasible due to the capacity constraint Eq. (3) for $n_1$. However, this can be made feasible by evicting $o_1$ to $n_6$ at a penalty of 1. If both migrations are performed ($M_{41}^2$ followed by $M_{41}^2$) a better placement will be obtained for the application, with a benefit of 8 vs. the current placement.

### 4 HEURISTICS CONSIDERING EVICTIONS FOR ENABLING SINGLE OPERATOR MIGRATIONS

In this section, we propose heuristics that consider evictions that in turn enable a beneficial migration so that the cumulative benefit/penalty is positive.

#### 4.1 Single Path Algorithm (SP)

In this algorithm, each node iterates through the list of
locally hosted operators to find the one (if any) that is most beneficial to migrate to a neighboring node. Then, it sends to the respective destination a hosting request with the identifier of the operator to be migrated, its processing requirements and the benefit of the migration as per Eq. (2).

When a node receives a hosting request it checks if it has enough free processing capacity to host the operator in question, in which case it sends a positive reply. Else, the host considers one or more evictions (in increasing order of their penalty) until enough free processing capacity is secured (or the cumulative penalty outweighs the benefit of the request). Then, for each such eviction, a hosting request is issued carrying the remaining benefit (used to decide for more evictions downstream). If all replies are positive and the total penalty does not exceed the benefit, a positive reply is sent back to the node that issued the hosting request.

When a node responds positively to a hosting request, it reserves the processing resources required to host the operator in question, including the resources (still) being used for the operators that are to be evicted. The above ensures that it will be possible to perform the respective migration, if the node that issued the hosting request decides to proceed. The aforementioned reservations are cancelled when a node receives a negative reply.

Finally, to avoid races, an operator must not be considered for more than one migration or eviction processes simultaneously. Moreover, we limit the degree of “recursive” forwarding of hosting requests via a hop limit, specified by the node initiating the migration process. Note that the interested reader can find the pseudocode of this algorithm in [15].

**Time Complexity.** The computational complexity at the source node is $O(\delta \phi k)$; with $\delta$, $\phi$, and $k$ denoting the maximum degree of a node, the maximum degree of an operator, and the number of operators hosted by the source node, respectively. At destination node the computational complexity becomes $O(\delta \phi k + k \log (k) + \delta \phi k (k+1)/2)$ which is equivalent to $O(\delta \phi k^2)$. The term $\delta \phi k$ denotes the complexity of calculating the eviction penalty for each pair combination (hosted operator, neighboring node). For each operator to be evicted, we store only the best (in terms of penalty) neighboring node as a candidate hosting node. As a result, a list of $k$ operators must be sorted according to the above penalty, with this complexity being reflected by $k \log (k)$. The first factor $\delta \phi k$ of the term $\delta \phi k (k+1)/2$ represents the complexity that each time an operator is removed we have to update the the eviction penalty of the affected operators (up to $\phi$); while the last factor $(k+1)/2$ represents the fact that each time we perform those updates, the list decreases by one operator. Because operator evictions can take place at a depth of $h$ (where $h$ is the hop limit), the complexity $O(\delta \phi k^2)$ becomes $O(h \delta \phi k^2)$. The time complexity for a node to receive/send messages is $4ksD$, because it xend send/receive up to two messages to migrate an operator; with $s$ being the maximum number of segments a message can be split, while $D$ representing the maximum delay to transmit one segment between nodes of one hop distance. Putting them all together and noting that $D$ is constant, the total time complexity becomes $O(h \delta \phi k^2 + hks)$. The memory consumed by either source node protocol or destination node protocol is due to the matrix encoding data dependencies, which demands $O(k \phi)$ memory. The complexity for keeping the load for each hosted operator is $O(k \delta)$. The matrix encoding the placement of neighboring operators of the $k$ hosted operators demands $O(k \phi \delta)$ memory. All the above are dominated by $O(k \phi \delta)$.

**Space complexity.** The memory consumed by either source node protocol or destination node protocol is due to the matrix encoding data dependencies, which demands $O(k \phi)$ memory. The complexity for keeping the load for each hosted operator is $O(k \delta)$. The matrix encoding the placement of neighboring operators of the $k$ hosted operators demands $O(k \phi \delta)$ memory. All the above are dominated by $O(k \phi \delta)$.

### 4.2 Network Flooding Algorithm (FL)

In single path algorithm (SP) a node chooses to evict operators in increasing order of the respective penalty. However, the latter is calculated locally, without knowing what the actual penalty of such migrations will be (an eviction may lead to further evictions downstream). To address the problem, we propose an algorithm where the operator to be evicted is chosen based on the smallest “total” penalty of the action.

The main difference of FL compared to SP is that the algorithm determines the cost of an operator eviction by investigating all possible destinations; not just the most promising one according to local knowledge. More specifically, a so-called probe request is sent to each destination that is a candidate for hosting the operator to be evicted.

When all replies arrive, the one with the greatest benefit (smallest penalty) is selected and the corresponding node is appointed as the destination for the migration/eviction in question.

The probe replies travel back the same way hosting replies do, with the difference that a reply also includes, besides the cumulative penalty, the respective eviction list. Eventually, the node that started the process (issued the probe request for the beneficial migration) receives such a reply. If this is positive, a hosting request is sent downstream, else the migration is (silently) cancelled. Unlike in SP, a hosting request specifies the evictions to be performed, therefore a node knows what operator(s) has to evict to what nodes.

Unlike in SP, an operator may be considered for eviction in the context of different requests at the same time. This process is to reduce excessive “locking conflicts” that would occur due to the flooding nature of the algorithm.

More specifically, a host request can be issued for an operator that is already involved in a probe request for which no reply has been received yet. In other words, hosting requests have precedence over probe requests. However, to avoid having numerous races, that in turn may result in many failed hosting requests, a hosting request cannot concern an operator involved in another pending hosting request and a probe request cannot concern an operator involved in a pending probe or hosting request. We also note that probe replies not do guarantee capacity reservation.

As a consequence a node may receive a hosting request for an operator that is no longer hosted locally (in which case it sends a negative reply).

The interested reader can find an example for the functionality of FL in [15].

**Time and Space Complexity.** The time complexity of FL is...
similar with that of SP. The only difference is that the term $4kSD$ becomes $4ksD$, since a node sends/receives up to $2\delta$ messages for each hosted operator. On the other hand, the space complexity of FL is the same with that of SP.

5 Heuristic for Enabling Group Migration through Evictions (GME)

The algorithms presented so far consider only single operator migrations per hosting request. This is a drawback because feasible beneficial migrations may not be performed when co-located operators have strong data interdependencies. For the above reason, a new algorithm (called GME) is introduced to perform group operator migrations as well as operator evictions to free processing resources when the candidate hosting node cannot host the respective operator group due to processing resource constraints. The algorithm proposed is split into three stages.

At the first stage, the current hosting node identifies the group of operators to be migrated and sends a hosting request to the candidate hosting node. If the candidate hosting node has the required capacity to host the respective operator group, a positive reply is sent back and the operator group migration is performed. (The algorithm does not proceed with the second stage). Otherwise, the candidate hosting node considers whether the node in question has free processing resources (actually a fraction of the required processing resources) or can free processing resources by evicting operators. If nothing of the above holds, then a negative reply is sent back. Otherwise, the algorithm proceeds with the second stage.

At the second stage, an eviction matrix is created. The eviction matrix encodes all of the operators that can be evicted from the candidate hosting node along with their size, as well as the network overhead that will be incurred within the network after the respective evictions. The aforementioned information is recorded in the matrix in an aggregate fashion. After the construction of the eviction matrix, the candidate hosting node sends the eviction matrix to the current hosting node of the operator migration group. Next the algorithm proceeds at the third stage.

At the third stage, the current hosting node receives the eviction matrix. Clearly, the current hosting node must prune some operators from the operator migration group such that the new migration group has aggregate size less than or equal to the free capacity of the candidate hosting node plus the capacity that is to be freed by the operators to be evicted.

5.1 Identification of Operator Migration Group

GME is a completely different approach compared to SP and FP. GME considers 1-hop migrations in a group-wise fashion by taking into account the operator data dependencies. Conversely, SP and FP migrate individual operators. GME is also equipped with the same recording mechanism as that of FP and SP. Each node within the system running GME is able to identify disjoint application sub-trees hosted locally on the respective nodes (using also partial information of the application tree). For each identified sub-tree, an operator group is produced that may be a subset of the sub-tree. For each group, a single destination is chosen as a host for all of the operators that are part of the group. Initially, the algorithm performs group operator migrations, and in the sequel single operator migrations. The single operator migrations are identified in the same procedure as that of SP and FP. The group operator migrations are identified as follows:

Sub-tree Identification. First, one or more disjoint sets of communicating locally hosted operators (belonging to the same application) are identified. Each such set corresponds to a part of the application tree; hereafter, referred to as a sub-tree. Specifically, the sub-tree identification process begins as follows: (i) create a sub-tree rooted on a locally hosted operator not belonging to an already identified sub-tree and (ii) add to the respective sub-tree each locally hosted operator that is adjacent (according to the application graph) to one of the operators belonging already to that sub-tree. Repeat phase (ii) until no operator can expand the respective sub-tree. After the expansion of a sub-tree completes, repeat phase (i) and (ii) accordingly, until all of the operators have been considered.

Partial migration benefit calculation. For each operator ($o_k$ if the operator is rooted on the sub-tree, otherwise $o_0$) of the corresponding sub-tree the partial migration benefit value is computed as:

The partial migration benefit is calculated for each operator in a top-down fashion. The operator of the up-most level is called root. Eq. 4 is used to calculate the partial benefit of migrating the root or of a sub-tree from $ns$ to $nd$. The above partial benefit corresponds to the actual benefit of that migration, provided that the operators of the corresponding sub-tree communicating with or remain on $ns$. To calculate the partial migration benefit of every other operator $o_k$ of a sub-tree we make use of Eq. 5, given that the parent of $o_k$ is $o_m$ (in terms of the corresponding sub-tree). Specifically, Eq. 5 corresponds to the load impact if both $o_k$ and its parent $o_m$ (denoted by $p(o_k) = o_m$) migrate to $nd$, on the premise that the rest of the operators of the corresponding sub-tree that communicate with $o_k$ remain on $ns$. The third term of Eq. 5 is justified by the fact that when calculating the partial migration benefit of an operator, there is an assumption that the corresponding operator is migrated along with its parent. Note that $l^k_d$ intrinsically considers that $p(o_k)$ is not on $ns$ but on $nd$. Therefore, instead of considering the load between $o_k$ and $p(o_k)$ as a benefit to the partial migration benefit, it is considered as a burden. To correct the above, the traffic load between the respective operator and its parent is added twice to Eq. 5. The actual benefit for migrating any operator $o_k$ together with all its predecessors (in the path) up to the root or is equal to the sum of the respective partial migration benefit values. We must note that without loss of generality in our examples we do not use sinks.

Sub-tree contraction. The algorithm processes a sub-tree by performing iteratively in a bottom-up fashion
prunes/merges of the leaves of the corresponding sub-tree. Specifically, if the partial migration benefit of a leaf is negative, then the respective leaf is pruned. Otherwise, it is chosen to merge the leaf with its parent by aggregating their partial migration benefits. Each merge produces a so-called contracted node with a respective partial migration benefit. The sub-tree contraction phase terminates when either all of the tree nodes have been pruned or it results in a single contracted node (called final contracted node) that represents a group operator migration.

As an example, consider the application tree shown in Fig. 4, whereby operators are denoted in capitals and data sources in small case letters. Without loss of generality we ignore the sink. An edge value stands for the data dependencies between operators/data sources adjacent to the corresponding edge. Let the application be deployed on a network as illustrated in Fig. 5. The identified sub-tree hosted at n1 is $\{A, B, C, D, E, F, G, H, I, J, K, L, M\}$. However, as observed in the subsequent text, by performing group operator migrations we can result in a better placement than that of Fig. 5. Specifically, we illustrate in Fig. 6 the calculation of the partial migration benefits of the identified sub-tree. As stated previously, the above calculation takes place in a top-down fashion. Therefore, we first calculate the partial migration benefit $pb_{A,B}$ for the migration of operator A from n1 to n2. The above partial migration benefit equals to -8. The above benefit results from Eq. 4 (note that operator A is the root of the sub-tree under consideration). We should also note that for the next levels of the sub-tree under consideration, we calculate the sub-tree’s partial migration benefits by making use of Eq. 5. Specifically, for the second level of the tree, we calculate the partial migration benefits of operators B, C, and E that being -5, -3, and 3, respectively. In Fig. 7, Fig. 8, and Fig. 9, we show the tree contraction phases taking place in a bottom-up fashion. Because the partial migration benefit of the leaf H appearing in Fig. 6 is negative, H is pruned from the tree (Fig. 7). On the other extreme, we observe that the partial migration benefits of the rest leaves are not negative. Therefore we choose to merge each one with its parent (Fig. 7). We observe that the partial migration benefits of the new leaves $\{B, D\}, \{I, J, K\}, \{M, G\}, \{L, F\}$ are positive. Therefore, at the next phase, the leaves depicted in Fig. 7 are merged with their parents, with the result being shown in Fig. 8. At the last phase, the leaves shown in Fig. 8 are merged with their parents. The result of the above is shown in Fig. 9. As a result, a migrating group is formed $\{A, B, D, E, I, J, K, M, G, C, L, F\}$, with its impact (migration benefit) within the system equalling 42 data units.

5.2 Creating the Eviction Matrix

Even though at the previous stage the candidate hosting node didn’t manage to free the required processing resources to host the respective operator group; during its effort to free processing resources it examined which operators can be evicted as well as the network overhead that will be incurred within the network after the respective evictions. The aforementioned operators are kept in an eviction list sorted in ascending order according to the ratio (operator eviction network overhead)/(operator required resources). In that way, the top entry of the eviction list represents the operator incurring the least network overhead per operator’s processing resource unit. According to the aforementioned list, an $(k+1)\times 3$ eviction matrix is created in the following way. The first entry of the matrix represents the candidate hosting node, while the rest entries represent the $k$ operators located in the eviction list. On the other hand, the columns of the list represent the operator id or node id, the aggregate required resources, and the aggregate network overhead. We must note that the aggregate required resources include also the free processing resources on the candidate hosting node. For example, for the $k^{th}$ entry the aggregate resources equals the free resources of the candidate hosting node plus the cumulative operator resources of the operators located at the first $k$ entries of the eviction matrix. On the other hand, for the $k^{th}$ entry the aggregate network overhead equals the cumulative network overhead of the operators located at the first $k$ entries of the eviction matrix. Next, the first entry of the list is removed and the operator contained in it is placed on second entry of the matrix. Each time an entry is filled in the eviction matrix, the eviction list is updated in the following way. For each entry of the eviction list we examine whether the respective operator and the operator contained in the current filled entry of the matrix (i) are to be evicted towards the same node and (ii) have data dependencies. If both (i) and (ii) hold, the network overhead of the current entry in the eviction list is updated and the list is re-sorted. Otherwise, no update is performed. The aforementioned are performed until the list is empty.

For example, assume the deployment of the operators shown in Fig. 5 on the underlying network. Consider that...
n_3 has free processing resources of 20 units, while n_2 of 1 unit. The processing resources required by each of the operators A, B, C, D, E, F, G, H, I, J, K, and M is of 1 processing unit. Required resources of P, O, and N are of 2, 3, and 2 processing units, respectively. To migrate the group of operators (or part of them) shown in Fig. 9 from n_1 towards n_2, the latter node must evict operators. First, the eviction list is created in the following way. The ratio (eviction network overhead)/(required resources) for the operator P, O, and N is of 2/2, 5/3, and 13/2, respectively. Therefore, P, O, and N are placed on the first, second, and third entry of the eviction list, respectively. The first entry of the eviction matrix contains the current hosting node along with its required resources (1 processing unit) and the network overhead incurs (0 data units). Next, the first entry of the list is removed and the operator contained in it is placed on the second entry of the matrix. Note that the aggregate required resources of the second entry of the matrix equal three processing units (P’s required resources plus the available resources of n_2), while the network overhead equals 2 data units. Then, the list is iterated with the rest entries being updated (if possible). Specifically, the entry containing O is updated because both P and O are to be evicted on the same node and they communicate with each other. The updated ratio equals 4/4. On the other hand, the entry containing N is not updated because N and P do not have data dependencies. Next, the first entry of the list, which contains P, is removed and P is placed on the third entry of the matrix. The aggregate required resources for the third entry of the matrix equals 6 processing units (3+3), while the network overhead equals 4 data units (the network overhead incurred due to the migration of both P and O towards n_2). Afterwards, the single entry of the list (containing N) is updated, with the ratio being 9/2. Subsequently, the entry of the list is removed and P is placed on the fourth entry of the matrix. The aggregate required resources for the fourth entry of the matrix equals 8 processing units (6+2), while the network overhead equals 10 data units (the network overhead incurred due to the eviction of P, O, and N towards n_2).

5.3 Pruning Operators from the Operator Migration Group

At this phase, the current hosting node of the operator migration group receives an operator eviction matrix from the candidate hosting node. The current hosting node creates an operator migration matrix which is an extension of the operator eviction matrix. Specifically, the operator eviction matrix is extended with the following fields: (a) the operator migration group and (b) the migration benefit. The operator migration group of the kth entry represents a group of operators that are to be migrated on the candidate hosting node. It must be noted that the aggregate required resources by the operators contained in the group must be less than the aggregate free resources of the kth entry. The migration benefit of the kth entry represents the actual benefit of migrating the operator migration group of the kth entry towards the candidate hosting node minus the network aggregate network overhead.

Note that initially, the operator migration group of each entry of the operator migration matrix is NULL, while the migration benefit equals 0.

\[
ab_{sd}^k = \ell_s^k - \ell_d^k
\]  

(6)

**Actual migration benefit calculation.** For each operator o, of the sub-trees identified in the first stage, the actual migration benefit is computed by Eq. (6). The actual migration benefit of an operator represents the benefit in the network overhead when migrating the operator in question towards the candidate hosting node.

**Sub-tree contraction.** The algorithm processes all of the sub-trees identified at the first stage by performing iterative pruning/merges between the nodes for each sub-tree. A prune takes place in a bottom-up fashion only on negative leaves. A merge takes place (not necessarily in a bottom-up fashion) by grouping a sub-tree node (called child) with its parent sub-tree node (called parent). Specifically, the resultant group (called contracted node) consists of the operators represented by the child sub-tree node and the parent sub-tree node, respectively. The partial migration benefit of the resultant group equals the sum of the partial migration benefits of the child and its parent, respectively. The actual migration benefit of the resultant group equals the sum of the partial migration benefit of the child plus actual migration benefit of the parent. At each iteration, the following take place: (a) each leaf having negative partial migration benefit is pruned, (b) the merge (among all of the sub-trees) resulting in the greater actual migration benefit is performed. The sub-tree contraction phase terminates when there cannot be produced any merge due to either (i) the capacity violation of the aggregate resources of the last entry of the operator migration matrix, (ii) the fact that there is no other merge to be performed.

**Knapsack execution.** At each iteration of the sub-tree contraction phase, the knapsack algorithm is executed as many times as the entries of the operator migration matrix. When the knapsack is executed for the kth entry of the operator migration matrix, the knapsack size equals...
the aggregate resources of the entry in question. Each node of all of the identified sub-trees represents a knapsack object. The benefit of a knapsack object is the actual migration benefit of the corresponding tree-node. The size of a knapsack object is the aggregate required resources of all of the operators participating in the corresponding tree-node. Each time the knapsack is executed, the total benefit of the knapsack (knapsack benefit) is recorded. The aggregate network overhead of the corresponding matrix entry is subtracted from the knapsack benefit. If the resultant benefit is less than or equal to the migration benefit of the corresponding matrix entry, then the process continues with the next knapsack execution (if any). Otherwise, the following take place. The operator migration group of the entry in question is updated to consist of all of the operators participating in the knapsack solution. The migration benefit of the entry in question is updated to the resultant benefit.

<table>
<thead>
<tr>
<th>Table 1. Operator eviction matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator/node id</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>( n_2 )</td>
</tr>
<tr>
<td>( P )</td>
</tr>
<tr>
<td>( O )</td>
</tr>
<tr>
<td>( N )</td>
</tr>
</tbody>
</table>

Last, when the contraction phase terminates along with the corresponding knapsack executions, the entry with the greater migration benefit is chosen. In case of a tie, the entry with the lesser size of the operator migration group is chosen. Next a new request is sent back to the candidate hosting node along with the chosen operator migration group as well as the operators to be evicted.

As an example, consider the deployment of the application tree shown in Fig. 4 on the network shown in Fig. 5. A request for the operator migration group depicted in Fig. 9 is sent to \( n_2 \). Because \( n_2 \) does not have the required resources to host the operator group in question, the operator eviction matrix shown in Table 1 is sent back to \( n_1 \). Next, \( n_1 \) constructs the operator migration matrix shown in Table 2 and computes the actual migration gains as shown in Fig. 10. As observed, the actual migration benefit for \( H \) is not computed. The above is because \( H \) is to be pruned due to its negative partial migration gain. Due to the fact that there is no tree-node with actual migration benefit greater than zero, it is meaningless to execute the knapsack algorithm. After pruning \( H \) and merging \( L \) and \( F \), we result in Fig. 11. Note that the merge of \( L \) and \( F \) leads to the greatest actual migration benefit, which is of 7 data units.

As shown in Fig. 11, there is only one tree-node \([F, L]\) having actual benefit greater than zero. Therefore, there is no reason to run knapsack. However, the operator eviction matrix must be updated. Specifically, the first entry remains as is because the group \([F, L]\) is of two resource units and the aggregate resources of first entry are only of 1 unit. The operator migration group of the second entry is updated to \([F, L]\), while the migration benefit is updated to 5 data units (7-2). The same holds for the third entry, with the migration being updated to 3 (7-4). The last entry is not updated because we result in a migration benefit that is less than zero.

<table>
<thead>
<tr>
<th>Table 2. Operator migration matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator/node id</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>( n_2 )</td>
</tr>
<tr>
<td>( P )</td>
</tr>
<tr>
<td>( O )</td>
</tr>
<tr>
<td>( N )</td>
</tr>
</tbody>
</table>

The next most beneficial merge is that of merging \( M \) and \( G \) (shown in Fig. 12), resulting in an actual migration benefit of 4 data units. Running knapsack for all of the entries of the migration matrix, we result in the following: (a) the operator migration group of the third entry is updated to \([F, L, M, G]\), while the migration benefit of the respective entry becomes 7 data units (7+4-4); and (b) the operator migration group of the fourth entry is updated to \([F, L, M,G]\), while the migration benefit of the respective entry becomes 1 data unit (7+4-10).

The merge of \( I \) and \( J \) (shown in Fig. 13) is the next most beneficial, resulting in an actual migration benefit of 3 data units. Running knapsack for all of the entries of the operator migration matrix, we result in the following: (a) the operator migration group of the fourth entry is updated to \([F, L, M, G, I, J]\), while the migration benefit of the respective entry becomes 8 data units (7+4+3-6); (b) the operator migration group of the fourth entry is updated to \([F, L, M, G, I, J]\), while the migration benefit of the respective entry becomes 4 data units (7+4+3-10).

The next most beneficial merge is that of merging \( I, J \) and \( K \), resulting in an actual migration gain of 14 data units. The above is shown in Fig. 14. Running knapsack for all of the entries of the operator migration matrix, we result in the following: (a) the operator migration group of the second entry is updated to \([I, J, K]\), while the migration benefit of the respective entry becomes 12 data units (14-2); (b) the operator migration group of the third entry is updated to \([I, J, K, F, L]\), while the migration benefit of the respective entry becomes 17 data units (14+7-4); and (c) the operator migration group of the fourth entry is updated to \([I, J, K, F, L, M, G]\), while the migration benefit of the respective entry becomes 15 data units (14+7+4-10).

The next most beneficial merges are: (a) the merge of \( B \) and \( D \) shown in Fig. 15, resulting in an actual migration gain of 3; (b) the merge of \([B, D]\) and \( A \) shown in Fig. 16, resulting in an actual migration gain of 3; (c) the merge of \([A, B, D]\) and \( E \) shown in Fig. 17, resulting in an actual migration gain of 6; (d) the merge of \([A, B, D, E]\) and \([I, J, K]\) shown in Fig. 18, resulting in an actual migration gain of 24; (e) the merge of \([F, L]\) and \( C \) shown in Fig. 19, resulting in an actual migration gain of -2. Running knapsack for all of the above merges, we result in no updates.
Note that there cannot be any new merge, because it will lead to the violation of the aggregate resources of the last matrix entry. As observed at Table 3, the most beneficial operator migration group is \{I, J, K, F, L\}, with the migration benefit being 17. Therefore, \( n_1 \) sends a new request towards \( n_2 \) about the operator migration group \{I, J, K, F, L\} as well as the operator eviction group \{P, O\}.

Table 3. operator migration matrix after sub-tree contraction

<table>
<thead>
<tr>
<th>Operator node id</th>
<th>aggr. resources</th>
<th>aggr. network overhead</th>
<th>operator migration group</th>
<th>migr. benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_2 )</td>
<td>1</td>
<td>0</td>
<td>{NULL}</td>
<td>0</td>
</tr>
<tr>
<td>( P )</td>
<td>3</td>
<td>2</td>
<td>{I,J,K}</td>
<td>12</td>
</tr>
<tr>
<td>( O )</td>
<td>6</td>
<td>4</td>
<td>{I,J,K,F,L}</td>
<td>17</td>
</tr>
<tr>
<td>( N )</td>
<td>8</td>
<td>10</td>
<td>{I,J,K,F,L,M,G}</td>
<td>15</td>
</tr>
</tbody>
</table>

Therefore, GME results in a placement with network overhead of 35 (48+17+4) abstract units. Note that 48 stands for the initial network overhead, 17 represents the benefit of the operator migration group, while 4 reflects the penalty of the operator eviction group. On the other extreme, SP and FL do not perform any migration, with their network overhead being 48.

Time Complexity. At source node we have: (a) the sub-tree identification phase with its complexity being \( O(\delta k) \); and (b) the contraction phase, with the complexity of each contraction being \( O(k \cdot \log(k) + k\nu) \) which is equivalent to \( O(k\nu) \), where \( (i) \) \( k \) stands for the computational cost of updating the actual and partial benefits. (ii) \( \log(k) \) represents the cost of re-sorting the sorted list of the contracted nodes due to the new contraction. Note that the list of contracted nodes is sorted in descending order according to the ratio [actual benefit of the contracted node]/[size of the contracted node]. And (iii) \( k\nu \) stands for performing knapsack by placing into knapsack each time the head of the list of contracted nodes, with \( \nu \) denoting the number of rows of the eviction matrix. Since there can be at most \( k-1 \) contractions and \( \delta \) candidate destination nodes, the total complexity of the contraction phase becomes \( O(\delta k^2\nu) \). At destination node, the complexity of creating the eviction matrix is \( O(\delta k\phi + k\log(k)) + (\delta k\phi + k\nu)(k + 1)/2 \), which is equivalent to \( O(\delta k^2\phi) \). Note that the first two terms are the same as those of SP. The first factor of third term reflects the complexity that each time an operator is removed we have to update the eviction penalty of the affected operators (up to \( \phi \)), as well as to re-sort those affected operators; while the last factor represents the fact that each time the above updates and re-sorts take place, the eviction list size decreases by one. Regarding the complexity of sending/receiving messages, as well as how the complexity is affected by the hop limit \( (h) \), the reader is referred to the corresponding part of SP.1 Putting them all together, the total complexity of GME becomes \( O(\delta k^2\nu + \delta h k^2\phi + \bar{h} k\nu) \).

Space Complexity. The space complexity of GME is similar with that of SP, with the difference being the extra space complexity \( (a) \) \( O(k) \) to store in memory a subtree, \( (b) \) \( O(k) \) for the sorted list of contracted nodes; and \( (c) \) \( O(k^2) \) for the eviction matrix. The dominating factor is \( O(k^2) \).

As we can observe, GME is scalable regarding both time and space. It is worth noting that space complexity is not affected by the network size, while time complexity is linearly affected by the degree of nodes and not at all by the network size.

6 NP-COMPLETENESS, CONVERGENCE, AND CONTROL MESSAGE SILENCE MECHANISM

This section discusses through rigorous analysis that: \( (a) \) the problem is NP-Complete, \( (b) \) the proposed algorithms result in convergence, and \( (c) \) after convergence is achieved, algorithms stop exchanging control messages.

6.1 NP-Completeness

Here we discuss the NP-complete reduction of the problem discussed in this paper.

**Theorem 1.** The operator placement problem is NP-complete given that the nodes within system have limited processing resources.

**Proof.** Assume a knapsack instance with \( K \) objects \( q_k \) (\( 1 \leq k \leq K \)) where \( s_k \) and \( \nu_k \) denote the size and value of \( q_k \) respectively. The knapsack problem consists of finding the collection of objects of maximum total value that fits in the knapsack of size \( S \). We can transform any such
statement to an equivalent statement of the operator placement problem studied in this paper as follows. The application tree consists of an operator \( o_k \) at the first level of the tree, which exchanges data with \( K \) operators \( o_k \) (\( 1 \leq k \leq K \)) found at the second level of the tree, corresponding one to one to the knapsack objects. At the second level, \( K \) data sources \( d_k \) exist (\( 1 \leq k \leq K \)), so that each operator belonging to the second level accesses data from exactly one data source. The communication overhead between \( o_k \) and the rest operators is set to \( e \), where \( e \leq \min(v_k) \), and between \( o_k \) and \( d_k \) is set to \( v_k - e \). Two nodes exist in the network \( n_1 \) and \( n_2 \). All operators are initially hosted at \( n_1 \), while \( n_2 \) holds all data sources together with \( o_k \). The processing requirements of \( o_k \) (\( 1 \leq k \leq K \)) is set to the corresponding knapsack’s object size \( s_k \), the processing requirements of \( o_k \) are set to \( 1 + \sum s_k \). Finally, the processing capacity of \( n_1 \) is set to \( \sum v_k s_k \), i.e., just enough to hold the operators hosted there, while the capacity of \( n_2 \) is set so that \( S \) free capacity remains. In the constructed operator placement problem instance, the network load is due to the communication of operators hosted at \( n_1 \) with \( o_k \) and the data sources hosted at \( n_2 \). The total network load of this placement is \( \sum v_k (v_k - e) + \sum e = \sum v_k v_k \). The only migrations that can be considered to minimize the load involve operators moving from \( n_1 \) to \( n_2 \) (data sources cannot move and the processing requirements of \( o_k \) are greater than the capacity of \( n_1 \)). It is easy to see that each migration of \( o_k \) from \( n_1 \) to \( n_2 \), decreases the network cost by \( v_k \) and can only be done provided that the free space \( S \) at \( n_2 \) is not covered. Thus, a solution to the aforementioned operator placement problem instance provides a solution to the initial knapsack instance.

6.2 Convergence

In this section we prove the convergence of the proposed algorithms.

Theorem 2. The proposed algorithms converge always to a stable placement.

Proof. It is important to note that a beneficial migration as per Eq. (2) is guaranteed to lead to a better placement only if operators having data dependencies (in the application tree) are not allowed to change hosts concurrently. Otherwise, it would be possible to have a never ending loop of “swaps”. The algorithms can be easily extended to satisfy the aforementioned constraint, e.g., by notifying the relatives of an operator before commencing with the actual migration process.

Given that our algorithms are equipped with the mechanism of preventing swaps, then each migration along with possible evictions must reduce the application-level traffic over the network. Specifically, the algorithms decide for one or more evictions in the context of a beneficial migration, only if the series of migrations and evictions will reduce the total network load by at least 1. Assuming a stable communication pattern between the operators, totaling \( x \) data units per time unit, at most \( x \) beneficial migrations can take place. While each beneficial migration may trigger a number of evictions, this number of evictions is also bounded by the network diameter (there are no cycles). The total number of migrations is bounded and, eventually, there will be no more migrations or evictions to perform.

6.3 Control Message Silence Mechanism

Here we discuss a control message silence mechanism along with a proof that each algorithm results in a placement where no control messages are exchanged. Mechanism. Both algorithms are extended with a mechanism that stops the respective protocols from generating messages ad infinitum once convergence is reached. The process works as follows: (a) each time a negative reply is sent to a node, the node is added to an update list; (b) when a node receives a negative reply, it adds the sender to a block list (blocked nodes are not considered as candidates for probe and hosting requests); (c) when a node frees capacity (due to the migration of a local operator to a remote node), it sends an update message to each node in the update list, and clears the list; and (d) when a node receives an update message, it removes the sender from the block list, and forwards the update to the neighbors.

Theorem 3. Adopting the aforementioned mechanism, the proposed algorithms result in a stable placement they stop sending control messages.

Proof. Due to convergence, eventually, operator migrations will stop. The source(s) of the last migration(s) will issue update messages due to the hosting capacity freed locally, triggering the generation of host/probe requests at other nodes. But given that convergence has been reached, no more migrations can be decided. Therefore, each node from which a hosting/probe request originated will receive a negative reply, and will suppress the generation of new requests due to the blocking policy. When the final communication phase is over, there are no nodes that can generate any new update messages or hosting/probe requests, the control message silence is achieved.

7 Evaluation

To study the performance of the proposed algorithms, we conducted experiments using NS-2 [20]. The details of the simulation setup and the results of the experiments are discussed in the next subsections.

7.1 Setup

Network generation. Two classes of networks are used, with 50 and 20 nodes placed randomly in a plane of 120\times120 and 80\times80 distance units, respectively. In both cases, two nodes are assumed to be in range of each other if the Euclidean distance is less than 30 units. The corresponding tree-based routing topology is obtained by constructing a spanning tree. Ten different topologies are generated for each network class.

Application generation. The application tree structure is also generated randomly, based on an initial (given) number of data sources. Three different application struc-
tures are generated this way, with (50, 20), (25, 10) and (10, 5) (operators, data sources), referred to as app-50, app-25 and app-10, respectively. The operators are assigned from 5 to 100 units (uniformly distributed). The initial placement of operators, data sources, and sinks on the network is random. Nodes are initialized to have exactly the processing resources required to host the operators that have been assigned.

7.2 Reference algorithms

As a reference for the results achieved by SP, FL, and GME, we run DBA [9], ILA(S) and ILA(G) algorithms [13]. Specifically, DBA is based on a naïve mechanism for handling capacity constraints on nodes, performing only swaps between VMs in case a VM migration cannot be implemented due to capacity violation. On the other extreme, ILA(S) chooses to perform only single operator beneficial migrations, in the same way a beneficial migration is decided in the SP and FL algorithms. ILA(S) does not have a mechanism for notifying nodes when capacity is freed. Instead, with a certain probability each neighboring node is optimistically assumed to have enough free processing resources. Unfortunately, ILA(S) never achieves control message silence; even though it is guaranteed to converge, i.e., stop performing migrations. On the other hand, ILA(G) behaves in the same way as ILA(S), with the difference being that ILA(G) performs group operator beneficial migrations. The way of forming a group operator migration is the same as the one described in Section 5.1.

We also employ an exhaustive algorithm that computes the best placement, by starting from an unoccupied network and trying out all possible combinations of operators on nodes. However, the placement obtained this way may not be actually feasible, because it may be impossible to reach from the initial placement by performing a series of eligible operator migrations and evictions, due to the capacity constraint as present in Eq. (3). Thus the corresponding network overhead on what could be achieved even by an optimal algorithm.

7.3 Experiments

In a first set of experiments, we compare the placements obtained for the 20-node networks and one app-10 application, the initial processing capacity of the nodes increases to 1-4 times the average operator required resources in the system. We report the average results for the ten different network topologies and five different initial placements for each topology (i.e., 50 runs). No large variances were recorded.

Fig. 20 illustrates the load reduction vs. the initial placement achieved by the algorithms. We found that DBA had by far the worst performance against the rest algorithms due to its naive mechanism handling capacity constraints. Its load reduction ranged between 3% and 20%. To avoid cluttering, we do not show its results. GME achieved the best performance as compared to the rest algorithms (excluding exhaustive). It is also observed, the superiority of SP over ILA(S), ILA(G), FP, and DBA when the surplus capacity ranged between 1 and 3 times of the average operator required resources. On the other hand, when the surplus capacity equaled four times of the average operator required resources, ILA(G) achieved the second best performance.

The above is because, when enough extra resources are available, the probability of performing evictions decreases. Therefore, SP and FP behaves similarly with ILA(S), while GME similarly with ILA(G). Moreover, when the extra free capacity is (just) 2 times the average operator required resources, SP, FL, and GME algorithms perform close to the exhaustive algorithm. Particularly, the exhaustive algorithm results in 14%, 11%, and 9% better performance as compared to FL, SP, and GME, respectively; a very positive sign as to their effectiveness. When nodes have considerable free capacity GME and the exhaustive algorithm achieve practically equally good placements, a trend observed throughout all our experiments. On the other hand, SP, FL, ILA(S), and ILA(G) achieve also nearly the same performance. The aforementioned is natural since the probability of a node becoming the bottleneck for beneficial migrations drops with increasing free capacity. Therefore, good placements can be reached without (any) operator evictions.

For the next experiments, we ran the algorithms in the 50-node networks where a mix of fifteen applications (five app-50, five app-25 and five app-10 applications) were deployed. This time we increased the free capacity of each node by 2, 5, 10, 20, 40 and 80 times the average operator required resources. We did not run the exhaustive algorithm due to its prohibitive running time complexity. We also did not show DBA due to its bad performance. However it is worth mentioning that GME reduced network overhead against DBA by 75% in certain scenarios.

Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.
The trend in Fig. 21, is similar to the one that was observed in the small-scale experiment. However, the network overhead reduction of GME vs. ILA(S) and ILA(G) becomes impressive when the extra free capacity is tight. Specifically, the reduction of GME vs. ILA(S) ranged between 32% and 56%, while the reduction of GME vs. ILA(G) ranged between 32% and 45%. The superiority of GME vs. FL and SP is attributed to the fact that GME considers grouped migrations by employing the knapsack algorithm, while FL and SP consider only single operator migrations.

On the other hand, the inferiority of FL vs. SP is attributed to the contention introduced by the flooding mechanism. In a large-scale system, it is very likely that several migrations and evictions will be attempted concurrently, that in turn leads to a large number of conflicts, where beneficial migrations are hindered by less beneficial ones (including evictions). Moreover, given that each such conflict leads to the generation of negative replies, the control message silence mechanism may be activated prematurely, missing opportunities for migrations/evictions.

The ability of SP to perform a larger number of migrations (and evictions) than FL is clearly shown in Fig. 22, which plots the number of migrations/evictions performed per operator in the system. The difference between SP and FL was more pronounced when capacity was tight, which was also the case when SP performed notably better than FL. As free node processing capacity increased, the number of beneficial migrations that could be performed without having to do any evictions, thus all algorithms performed a comparable number of migrations (and SP started performing fewer migrations in total as the number of evictions dropped). ILA(S) performed the smallest number of migrations, with ILA(G) following closely when the available processing capacity was scarce. The above is justified by the fact that ILA(S) and ILA(G) were not equipped with the eviction mechanism. The above acted as a suppressing factor for both ILA(S) and ILA(G) for performing beneficial migrations. On the other hand, GME performed the greatest number of migrations, which is justified by the fact that GME resulted in more opportunities to migrations/evictions due to the consideration of grouped evictions with the contribution of knapsack algorithm. It must also be noted that the number of migrations performed by ILA(G) approaches the number of migrations performed by GME when the available processing capacity is 80 times the average operator processing requirements. The above is due to the fact that when the available processing capacity became big, the evictions were reduced, leading GME to perform in a similar way as ILA(G).

Fig. 23 shows the ratio of control messages to the number of migrations performed. Clearly, GME is more efficient than the rest algorithms. The above is because GME saves control messages when sending a hosting request for a whole group of operators. It is noteworthy to highlight that when nodes have little free capacity GME and SP results in almost the same number of control messages. The above is justified by the fact that GME did not have many opportunities to migrate group of operators when the available processing capacity was tight. On the other extreme, ILA(G) was close to GME when the surplus capacity was big. The above is due to the fact that the behavior of GME is similar with that of ILA(G) when the available processing capacity is big. The biggest per-migration protocol overhead was achieved by FL, which is due to the fact that FL algorithm flooded the network with probe requests and replies to find the best possible series of evictions, whereas SP and GME algorithms picked a single path.

The high per-migration overhead of ILA(S) and ILA(G) when the free capacity was tight, is justified by the fewer migrations accomplished compared to SP, FL, and GME algorithms. This overhead is clearly visible when available processing capacity is tight.

In a final set of experiments, we measured the impact of limiting the hops of hosting and probe requests in SP, FL, and GME algorithms. We used the 50-node networks and application mix of the previous experiments, while fixing the extra available node processing capacity at 10 times the average operator processing requirements. The load reduction achieved, the number of migrations per operator and the number of control messages per migration are depicted in Figs. 24, 25 and 26, respectively, with the hop limit varying from 1 to 8. The behavior of ILA(S) and ILA(G) was not affected by this parameter (the algorithm only issued 1-hop requests for beneficial migrations).
The algorithms SP, FL, and GME exhibited a similar performance for small hop limits. As the hop limit increases, GME clearly outperforms all of the algorithms. It is also observed that SP algorithm outpaced FL algorithm, due to the growing negative effects of flooding (and contention). It is interesting to observe that the load reduction achieved by SP and GME flattens at 4 hops being practically identical to the reduction achieved at 8 hops, despite the larger number of migrations (and evictions) performed in the latter case. This is attributed to the fact that, from a certain point onwards, additional evictions did not lead to a significantly better application placement. More specifically, the average diameter of the 50-node networks used in our simulations is 10. Moreover, a hop limit of 4 was already sufficient for a node that was not located at the periphery of the network to reach almost all other nodes. The protocol overhead of SP and GME starts dropping at 4 hops and this trend continued at 8 hops. The reason is that there were fewer opportunities to perform migrations (and evictions) when the hop limit was small, while the protocol overhead was amortized as the number of migrations grow at larger hop limits. On the other hand, the per-migration overhead of FL algorithm increased steadily due to the scalability problems of the flooding approach.

7.3.1 Time Performance Test cases

At this section, we show the response times for only GME, since this is the algorithm of interest in our paper. Specifically, because GME has been implemented in a fully distributed way and tested within a simulation environment, the total response time is not an important metric because it includes also the response time of the simulator. For this reason, we ran new experiments to analyze the response time of GME regarding a reference node that is to take a decision for the operator migration group as well as the operator eviction group. We tested the response times of GME at very large cases, whereby a node can handle up to 3800 operators.

For the first experiment, the number of operators hosted at the reference node ranges between 50 and 3800, while we keep fixed: (a) the degree of the reference node \( \delta = 5 \), and (b) the size of the eviction matrix at 50 operators. Regarding the second experiment we vary the reference node degree between 5 and 3800, while we keep fixed: (a) the number of operators hosted at reference node \( k = 200 \), and (b) the size of the eviction matrix at 50 operators. The results are shown at Table 4, where we applied the one-hop limit case. As observed, the scalability of GME is evident even for fictitious scenarios where the number of hosted operator at a node is greater than 50, or the node degree is greater than 50. We conducted also experiments for \( v \) and \( \varphi \); however, we do not show them because the results are similar with those of \( \delta \).

<table>
<thead>
<tr>
<th>Hop limit</th>
<th>Control msgs per migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FL: 8, SP: 12, ILA: 14, GME: 18</td>
</tr>
<tr>
<td>2</td>
<td>FL: 10, SP: 14, ILA: 16, GME: 20</td>
</tr>
<tr>
<td>4</td>
<td>FL: 12, SP: 16, ILA: 18, GME: 20</td>
</tr>
<tr>
<td>8</td>
<td>FL: 14, SP: 18, ILA: 20, GME: 22</td>
</tr>
</tbody>
</table>

Table 4. Response Times

<table>
<thead>
<tr>
<th>Hop limit</th>
<th>Control msgs per migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FL: 8, SP: 12, ILA: 14, GME: 18</td>
</tr>
<tr>
<td>2</td>
<td>FL: 10, SP: 14, ILA: 16, GME: 20</td>
</tr>
<tr>
<td>4</td>
<td>FL: 12, SP: 16, ILA: 18, GME: 20</td>
</tr>
<tr>
<td>8</td>
<td>FL: 14, SP: 18, ILA: 20, GME: 22</td>
</tr>
</tbody>
</table>

Figure 26. Control msgs vs. hop limit (50 nodes, cap +10, app-mix).

8 CONCLUSIONS

We have described distributed algorithms for migrating operators in big data environments to reduce application-level network traffic. Our approach introduces migrations that may be non-beneficial on their own, but free processing resources to enable beneficial migrations, which eventually leads to an overall better application placement on nodes. We showed that the problem in this paper is NP-complete, as well as that the proposed solutions converge. We also presented and discussed the results of extensive simulations, showing that the proposed approach outperforms solutions based solely on beneficial migrations, resulting in placements that reduce network traffic significantly. Our future plans include devising algorithms for general-structured networks.

ACKNOWLEDGMENT

This work is supported by the China National Basic Research Program (973 Program, No. 2015CB352400), NSFC under grant U1401258 and 61550110250.

REFERENCES


Copyright (c) 2016 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.


**AUTHOR’S BIOGRAPHIES**

Nikos Tziritas received his Ph.D. degree from the University of Thessaly, Greece, 2011. He is currently a research assistant professor in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research has been on scheduling, load-balancing and replication in CDNs, as well as energy optimization and resource management in WSNs and cloud computing systems. His work has appeared in over 25 publications.

Thanassis Loukopoulos received his Diploma in Computer Engineering and Informatics from the University of Patras, Greece, in 1997. He was awarded a PhD degree in Computer Science by the Hong Kong University of Science and Technology (HKUST) in 2002. Currently, he is lecturer in University of Thessaly, Greece. He had the best paper award in ICPP 2001. His current research interests include: green and cloud computing, distributed system optimizations, wireless networks.

Samee U. Khan is Associate Professor of Electrical and Computer Engineering at the North Dakota State University, Fargo, ND, USA. Prof. Khan’s research interests include optimization, robustness, and security of: cloud, cluster and big data computing, social networks, wired and wireless networks, power systems, smart grids, and optical networks. His work has appeared in over 300 publications. He is a Fellow of the Institution of Engineering and Technology (IET, formerly IEE), and a Fellow of the British Computer Society (BCS).

Cheng-Zhong Xu received the PhD degree in computer science from the University of Hong Kong in 1993. He is currently a professor in the Department of Electrical and Computer Engineering of Wayne State University, and the director of Cloud and Internet Computing Laboratory (CIC) and Sun’s Center of Excellence in Open Source Computing and Applications (OSCA). His research interest is mainly in scalable distributed and parallel systems and wireless embedded computing devices. He has published two books and more than 160 articles in peer-reviewed journals and conferences in these areas. He is an IEEE fellow.

Albert Y. Zomaya is the Chair Professor of High Performance Computing & Networking at Sydney University. Professor Zomaya published more than 500 scientific papers and articles and is author, co-author or editor of more than 20 books. He is an associate editor for 22 leading journals. Professor Zomaya is the recipient of the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), the IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), and the IEEE Computer Society Technical Achievement Award (2014). He is a Fellow of AAAS, IEEE, IET.